Express 工作原理和路由源码分析

Express 是一基于 Node 的一个框架，用来快速创建 Web 服务的一个工具，为什么要使用 Express 呢，因为创建 Web 服务如果从 Node 开始有很多繁琐的工作要做，而 Express 为你解放了很多工作，从而让你更加关注于逻辑业务开发。举个例子：

创建一个很简单的网站：

1. 使用 Node 来开发：

```
var http = require('http');
var url = require("url");

http.createServer(function(req, res) {
    res.writeHead(200, {
        'Content-Type': 'text/plain'
    });
    var url_str = url.parse(req.url,true);
    res.end('Hello World\n' + url_str.query);
}).listen(8080, "127.0.0.1");
console.log('Server running at http://127.0.0.1:8080/');
```

这是一个简单的 hello world，运行以后访问 http://127.0.0.1 会打印相关字符串，这是最普通的页面，但实际上真正的网站要比这个复杂很多，主要有：

（1） 多个页面的路由功能

（2） 对请求的逻辑处理

那么使用 node 原生写法就要进行以下处理

```
// 加载所需模块
var http = require("http");

// 创建 Server
var app = http.createServer(function(request, response) {
  if(request.url == '/'){
    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("Home Page!\n");
  } else if(request.url == '/about'){
    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("About Page!\n");
  } else{
    response.writeHead(404, { "Content-Type": "text/plain" });
```

```
    response.end("404 Not Found!\n");
  }
});

// 启动 Server
app.listen(1984, "localhost");
```

代码里在 createServer 函数里传递一个回调函数用来处理 http 请求并返回结果，在这个函数里有两个工作要做：

（1）路由分析，对于不同的路径需要进行分别处理

（2）逻辑处理和返回，对某个路径进行特别的逻辑处理

在这里会有什么问题？如果一个大型网站拥有海量的网站（也就是路径），每个网页的处理逻辑也是交错复杂，那这里的写法会非常混乱，没法维护，为了解决这个问题，TJ 提出了 Connect 的概念，把 Java 里面的中间件概念第一次进入到 JS 的世界，Web 请求将一个一个经过中间件，并通过其中一个中间件返回，大大提高了代码的可维护性和开发效率。

```
// 引入 connect 模块
var connect = require("connect");
var http = require("http");

// 建立 app
var app = connect();

// 添加中间件
app.use(function(request, response) {
    response.writeHead(200, { "Content-Type": "text/plain" });
    response.end("Hello world!\n");
});
// 启动应用 http.createServer(app).listen(1337);
```

但是 TJ 认为还应该更好一点，于是 Express 诞生了，通过 Express 开发以上的例子：

2. 使用 Express 来开发：

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});
app.get('/about', function (req, res) {
  res.send('About');
```

```
});
var server = app.listen(3000, function () { var host =
server.address().address; var port = server.address().port;
console.log('Example app listening at http://%s:%s', host, port); });
```

从 Express 例子可以看出，使用 Express 大大减少了代码函数，而且逻辑更为简洁，所以使用 Express 可以提高开发效率并降低工程维护成本。

首先 Express 有几个比较重要的概念：路由，中间件和模版引擎

开发人员可以为 Web 页面注册路由，将不同的路径请求区分到不同的模块中去，从而避免了上面例子 1 所说的海量路径问题，例如

```
var express = require("express");
var http = require("http");
var app = express();

app.all("*", function(request, response, next) {
    response.writeHead(404, { "Content-Type": "text/plain" });
    next();
});

app.get("/", function(request, response) {
    response.end("Welcome to the homepage!");
});

app.get("/about", function(request, response) {
    response.end("Welcome to the about page!");
});

app.get("*", function(request, response) {
    response.end("404!");
});

http.createServer(app).listen(1337);
```

开发人员可以为特定的路由开发中间件模块，中间件模块可以复用，从而解决了复杂逻辑的交错引用问题，例如

```
var express = require('express');
var app = express();

// 没有挂载路径的中间件，应用的每个请求都会执行该中间件
app.use(function (req, res, next) {
```

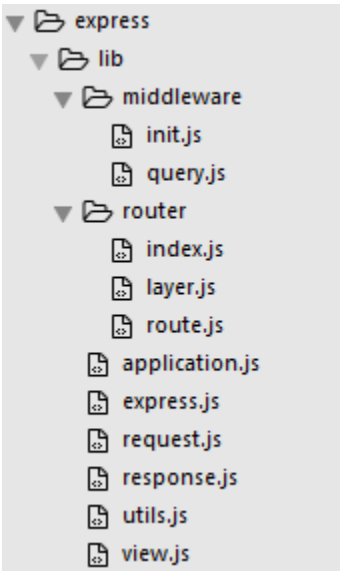
```
  console.log('Time:', Date.now());
  next();
});

// 挂载至 /user/:id 的中间件，任何指向 /user/:id 的请求都会执行它
app.use('/user/:id', function (req, res, next) {
  console.log('Request Type:', req.method);
  next();
});

// 路由和句柄函数(中间件系统)，处理指向 /user/:id 的 GET 请求
app.get('/user/:id', function (req, res, next) {
  res.send('USER');
});


var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;

  console.log('Example app listening at http://%s:%s', host, port);
});
```

同时 Express 对 Request 和 Response 对象进行了增强，添加了很多工具函数。

其中路由和中间件还有很多细节问题，可以参考 http://www.expressjs.com.cn/来学习

下面我们来看看 Express 的工作原理

我们首先来看看 Express 的源码结构：



简单介绍下：

Middleware:中间件

init.js 初始化 request，response

query.js 格式化 url，将 url 中的 rquest 参数剥离，储存到 req.query 中

Router:路由相关

index.js: Router 类，用于存储中间件数组

layer.js 中间件实体类

route.js route 类，用于处理不同 Method

Application.js 对外 API

Express.js 入口

Request.js 请求增强

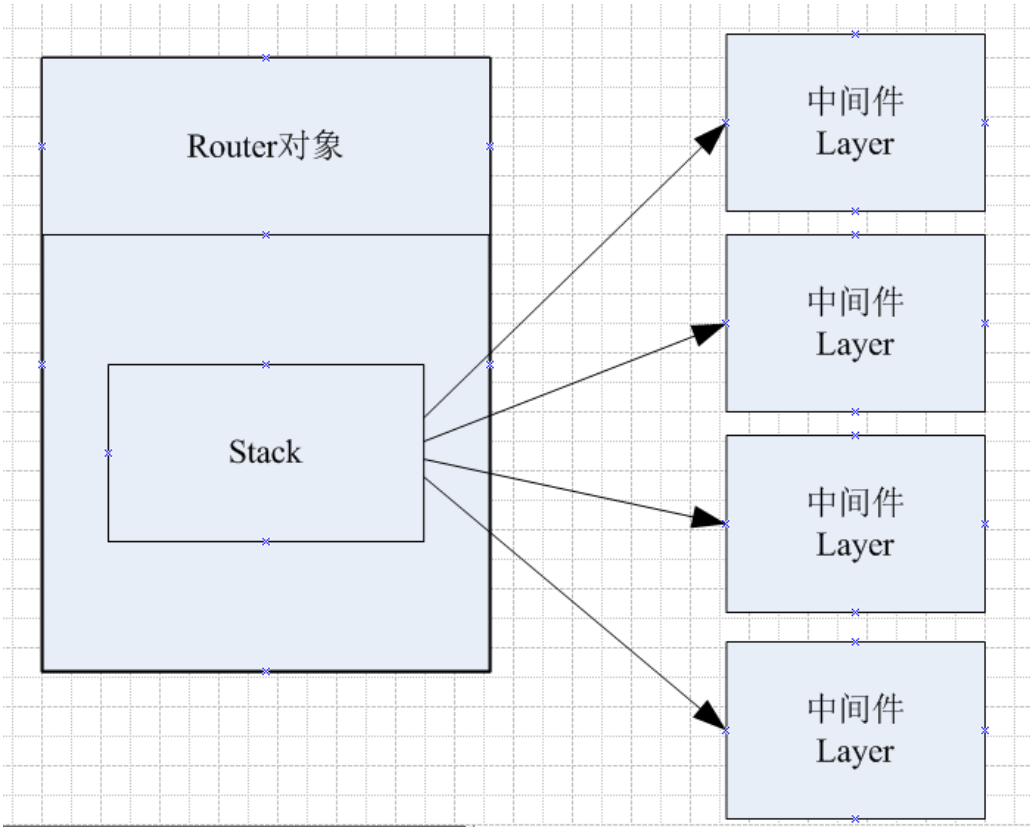
Response.js 返回增强

Utils.js 工具函数

View.js 模版相关

现在看不明白没关系，可以先看看后面的解释然后再回头看就明白了：

我们前面有说道路由和中间件，那么我们就需要有地方来保存这些信息，比如路由信息，比如中间件回调函数等等，express 中有一个对象 Router 对象专门用来存储中间件对象，他有一个数组叫 stack，保存了所有的中间件对象，而中间件对象是 Layer 对象。



Router 对象就是 router/index.js 文件，他的代码是：

```
var proto = module.exports = function(options) {
  var opts = options || {};

  function router(req, res, next) {
    router.handle(req, res, next);
  }

  // mixin Router class functions
  router.__proto__ = proto;

  router.params = {};
  router._params = [];
  router.caseSensitive = opts.caseSensitive;
  router.mergeParams = opts.mergeParams;
  router.strict = opts.strict;
  router.stack = [];

  return router;
};
```

Router 对象的主要作用就是存储中间件数组，对请求进行处理等等。

Layer 对象在 router/layer.js 文件中，**是保存中间件函数信息的对象**，主要属性有：

- handle，function，表示中间件函数
- name，string，中间件函数的名字，如果为匿名函数则为 <anonymous>
- params，undefined，在执行 match 的时候赋值
- path，undefiend，在执行 match 的时候赋值
- regexp，RegExp，路径的正则表达形式
- keys，[]，保存的是路径中的参数及其相关的一些其它信息
- route，如果是路由中间件，则该属性为一个 Route 对象，否则为 undefined。该属性不在 Layer 模块中定义，而是在 Router 模块中生成实例后定义

源码见：

```
function Layer(path, options, fn) {
  if (!(this instanceof Layer)) {
    return new Layer(path, options, fn);
  }

  debug('new %s', path);
  var opts = options || {};

  this.handle = fn;
  this.name = fn.name || '<anonymous>';
  this.params = undefined;
  this.path = undefined;
  this.regexp = pathRegexp(path, this.keys = [], opts);

  if (path === '/' && opts.end === false) {
    this.regexp.fast_slash = true;
  }
}
```

这里面的细节先不多考虑，只需要了解关键的信息 path,handler 和 route

handler 是保存中间件回调函数的地方，path 是路由的 url,route 是一个指针，指向 undefined 或者一个 route 对象，为何会有两种情况呢，是因为中间件有两种类型：

（1）普通中间件：普通中间件就是不管是什么请求，只要路径匹配就执行回调函数

（2）路由中间件：路由中间件就是区分了 HTTP 请求的类型，比如 get/post/put/head 等等（有几十种）类型的中间件，就是说还有区分请求类型才执行。

所以有两种 Layer，一种是普通中间件，保存了 name，回调函数已经 undefined 的 route 变量。

另外一种是路由中间件，除了保存 name，回调函数，route 还会创建一个 route 对象：

route 对象在 router/route.js 文件中，

```
function Route(path) {
  this.path = path;
  this.stack = [];

  debug('new %s', path);

  // route handlers for various http methods
  this.methods = {};
}
```

我们看到 route 对象有 path 变量，一个 methods 对象，也有一个 stack 数组，stack 数组其实保存的也是 Layer 对象，这个 Layer 对象保存的是对于不同 HTTP 方法的不同中间件函数（handler 变量）。

也许你会问，这个 route 的数组里面的 Layer 和上面 router 的数组里面的 Layer 有何不同，他们有一些相同之处也有一些不同之处，主要是因为他们的作用不同：

相同之处：他们都是保存中间件的实例对象，当请求匹配到指定的中间件时，该对象实例将会触发。

不同之处：

Router 对象的 Layer 对象有 route 变量，如果为 undefined 表示为普通中间件，如果指向一个 route 对象表示为路由中间件，没有 method 对象。而 route 对象的 Layer 实例是没有 route 变量的，有 method 对象，保存了 HTTP 请求类型。

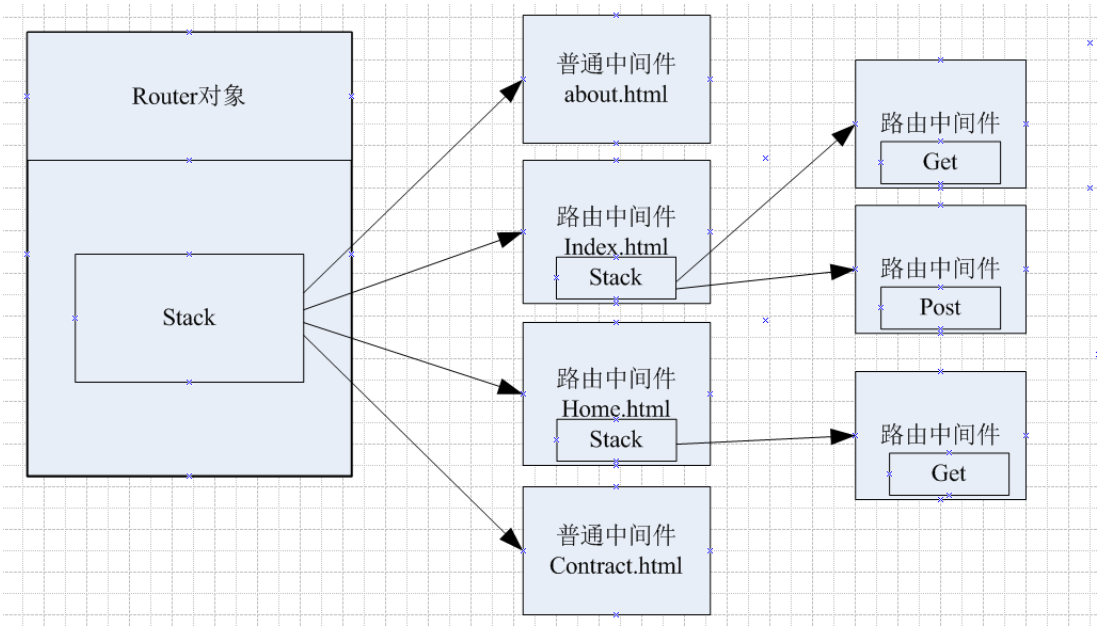
所以 Router 对象中的 Layer 对象是保存普通中间件的实例或者路由中间件的路由，而 route 对象中的 Layer 是保存路由中间件的真正实例。

我们来看个例子，加入有段设置路由器的代码：

app.use("/index.html",function(){ //此处省略一万行代码});

app.use("/contract.html",function(){ //此处省略一万行代码});
app.get("/index.html",function(){ //此处省略一万行代码});
app.post("/index.html",function(){ //此处省略一万行代码});
app.get("/home.html",function(){ //此处省略一万行代码});

代码中注册了 2 个普通中间件 about.html 和 contract.html，两个路由中间件，index.html 和 home.html，对 index.html 有 get 和 post 两种中间件函数，对 home.html 只有 get 中间件函数，在内存中存储的形式就是：



我们上面看到了几种注册中间件的方式，下面就来介绍下路由器的几个动作逻辑：

route 对象：

router.METHOD(path,callback);//METHOD 是 HTTP 请求方法(get/post 等),他的实现过程在这里：

```
methods.forEach(function(method){
  Route.prototype[method] = function(){
    var handles = flatten(slice.call(arguments));

    for (var i = 0; i < handles.length; i++) {
      var handle = handles[i];

      if (typeof handle !== 'function') {
        var type = toString.call(handle);
        var msg = 'Route.' + method + '() requires callback functions but got a ' + type;
        throw new Error(msg);
      }

      debug('%s %s', method, this.path);

      var layer = Layer('/', {}, handle);
      layer.method = method;

      this.methods[method] = true;
      this.stack.push(layer);
    }

    return this;
  };
});
```

methods 变量是一个数组包含了几十个 http 请求类型，这段代码给 route 对象添加了几十个方法，主要逻辑就是创建一个 Layer 对象，保存中间件函数对象和 Method 方法，添加到 route 的 stack 数组中去。

我们再来看看 Router 对象的方法：

```
proto.use = function use(fn) {
  var offset = 0;
  var path = '/';

  // default path to '/'
  // disambiguate router.use([fn])
  if (typeof fn !== 'function') {
    var arg = fn;

    while (Array.isArray(arg) && arg.length !== 0) {
      arg = arg[0];
    }

    // first arg is the path
    if (typeof arg !== 'function') {
      offset = 1;
      path = fn;
    }
  }

  var callbacks = flatten(slice.call(arguments, offset));

  if (callbacks.length === 0) {
    throw new TypeError('Router.use() requires middleware
functions');
  }

  for (var i = 0; i < callbacks.length; i++) {
    var fn = callbacks[i];

    if (typeof fn !== 'function') {
      throw new TypeError('Router.use() requires middleware function
but got a ' + gettype(fn));
    }

    // add the middleware
    debug('use %s %s', path, fn.name || '<anonymous>');

    var layer = new Layer(path, {
      sensitive: this.caseSensitive,
      strict: false,
      end: false
    }, fn);
```

```
    layer.route = undefined;

    this.stack.push(layer);
  }

  return this;
};
```

这个就是 app.use 的实现方法，实际上 app.use 就是调用了 router.use,后面详细介绍，先看看这个方法做了什么，当我们调用 app.use(function(){XXX});的时候，这里的函数首先判断了参数类型，看看有没有 path 传递进来，没有 path 就是"/"有的话保存到 path 变量，然后对后面的所有中间件函数进行了以下处理：

创建了一个 layer 对象，保存了路径，中间件函数并且设置了 route 变量为 undefined，最后把这个变量保存到 router 的 stack 数组中去，到此一个普通中间件函数创建完成，为何要设置 route 变量为 undefined，因为 app.use 创建的中间件肯定是普通中间件,app.METHOD 创建的才是路由中间件。

当我面调用 app.get("",function(){XXX})的时候调用的其实是 router 对象的 route 方法：

```
proto.route = function route(path) {
  var route = new Route(path);

  var layer = new Layer(path, {
    sensitive: this.caseSensitive,
    strict: this.strict,
    end: true
  }, route.dispatch.bind(route));

  layer.route = route;

  this.stack.push(layer);
  return route;
};
```

route 方法也创建了一个 layer 对象，但是因为本身是路由中间件，所以还会创建一个 route 对象，并且保存到 layer 的 route 变量中去。

现在我们总结一下：

1. route 对象的主要作用是创建一个路由中间件，并且创建多个方法的 layer 保存到自己的 stack 数组中去。

2. router 对象的主要作用是创建一个普通中间件或者路由中间件的引导着（这个引导着 Layer 对象链接到一个 route 对象），然后将其保存到自己的 stack 数组中去。

所以 route 对象的 stack 数组保存的是中间件的方法的信息（get，post 等等）而 router 对象的 stack 数组保存的是路径的信息（path）

好了，说完了这些基础组件，下面说一下真正暴露给开发者的对外接口，很显然刚才说的都是内部实现细节，我们开发者通常不需要了解这些细节，只需要使用 application 提供的对外接口。

- settings，{}，主要是一些设置信息，相关方法有：
  - set(setting, value)，有 value 的时候进行设置，无 value 的时候进行获取
  - get(setting)
  - enabled(setting)
  - disabled(setting)
  - enable(setting)
  - disable(setting)
- cache，{}
- engines，{}，设置模版引擎，相关方法有：
  - engine(ext, fn)
- locals，{}
- mountpath，字符串，相关方法有：
  - use(path, app)，会设置 app 的 mountpath 的值
  - path()，获取应用的绝对路径值
- _router，Router 对象，相关方法有（由于参数可以有多种形式，因此并未列出参数）：
  - route()，创建一条路由，会调用 Router.route
  - METHOD()，创建一条路由，并调用VERB方法，会调用 Router.route
  - all()，创建一条路由，并调用所有的VERB方法，会调用 Router.route
  - param()，会调用 Router.param
  - use()，会调用 Router.use

application 在 application.js 文件下，主要保存了一些配置信息和配置方法，然后是一些对外操作接口，也就是我们说的 app.use,app.get/post 等等，有几个重要的方法：

```
app.use = function use(fn) {
  var offset = 0;
  var path = '/';

  // default path to '/'
  // disambiguate app.use([fn])
  if (typeof fn !== 'function') {
    var arg = fn;

    while (Array.isArray(arg) && arg.length !== 0) {
      arg = arg[0];
    }

    // first arg is the path
    if (typeof arg !== 'function') {
```

```
      offset = 1;
      path = fn;
    }
  }

  var fns = flatten(slice.call(arguments, offset));

  if (fns.length === 0) {
    throw new TypeError('app.use() requires middleware functions');
  }

  // setup router
  this.lazyrouter();
  var router = this._router;

  fns.forEach(function (fn) {
    // non-express app
    if (!fn || !fn.handle || !fn.set) {
      return router.use(path, fn);
    }

    debug('.use app under %s', path);
    fn.mountpath = path;
    fn.parent = this;

    // restore .app property on req and res
    router.use(path, function mounted_app(req, res, next) {
      var orig = req.app;
      fn.handle(req, res, function (err) {
        req.__proto__ = orig.request;
        res.__proto__ = orig.response;
        next(err);
      });
    });

    // mounted an app
    fn.emit('mount', this);
  }, this);

  return this;
};
```

我们看到 app.use 在进行了一系列的参数处理后，最终调用的是 router 的 use 方法创建一个普通中间件。

```
methods.forEach(function(method){
  app[method] = function(path){
    if (method === 'get' && arguments.length === 1) {
      // app.get(setting)
      return this.set(path);
    }

    this.lazyrouter();

    var route = this._router.route(path);
    route[method].apply(route, slice.call(arguments, 1));
    return this;
  };
});
```

同 route 一样，将所有的 http 请求的方法创建成函数添加到 application 对象中去，从而可以使用 app.get/post/等等，最终的效果是调用 router 的 route 方法创建一个路由中间件。

所有的方法再通过 express 入口文件暴露在对外接口中去。而 middleware 中的两个文件是对 application 做的一些初始化操作，request.js 和 response.js 是对请求的两个对象的一些增强。